

## 6.001 SICP Variations on a Scheme

Beyond Scheme – designing language variants:

Lazy evaluation

- Complete conversion – normal order evaluator
- Upward compatible extension – lazy, lazy-memo

1/40

## Evaluation model

Rules of evaluation:

- If expression is self-evaluating (e.g. a number), just return value
- If expression is a name, look up value associated with that name in environment
- If expression is a lambda, create procedure and return
- If expression is special form (e.g. if) follow specific rules for evaluating subexpressions
- If expression is a compound expression
  - Evaluate subexpressions in any order
    - If first subexpression is primitive (or built-in) procedure, just apply it to values of other subexpressions
  - If first subexpression is compound procedure (created by lambda), evaluate the body of the procedure in a new environment, which extends the environment of the procedure with a new frame in which the procedure's parameters are bound to the supplied arguments

2/40

## Alternative models for computation

- Applicative Order:
  - evaluate all arguments, then apply operator
- Normal Order:
  - go ahead and apply operator with unevaluated argument subexpressions
  - evaluate a subexpression only when value is *needed*
    - to print
    - by primitive procedure (that is, primitive procedures are "strict" in their arguments)

3/40

## Applicative Order Example

```
(define (foo x)
  (write-line "inside foo")
  (+ x x))

(foo (begin (write-line "eval arg") 222))
=> (begin (write-line "eval arg") 222)
    => 222
=> (begin (write-line "inside foo")
         (+ 222 222))
```

Evaluating argument to foo

Evaluating second part of argument

From body of foo

We first evaluated argument, then substituted value into the body of the procedure

```
eval arg
inside foo
=> 444
```

4/40

## Normal Order Example

```
(define (foo x)
  (write-line "inside foo")
  (+ x x))

(foo (begin (write-line "eval arg") 222))
=> (begin (write-line "inside foo")
         (+ (begin (w-l "eval arg") 222)
            (begin (w-l "eval arg") 222))))
```

From body of foo

```
inside foo
eval arg
eval arg
=> 444
```

As if we substituted the unevaluated expression in the body of the procedure

5/40

## Normal order (lazy evaluation) versus applicative order

- How can we change our evaluator to use normal order?
  - Create "delayed objects" – expressions whose evaluation has been deferred
  - Change the evaluator to force evaluation only when needed
- Why is normal order useful?
  - What kinds of computations does it make easier?

6/40

## How can we implement lazy evaluation?

```
(define (l-apply procedure arguments env) ; changed
  (cond ((primitive-procedure? procedure)
        (apply-primitive-procedure
         procedure
         (list-of-arg-values arguments env)))
        ((compound-procedure? procedure)
         (l-eval-sequence
          (procedure-body procedure)
          (extend-environment
           (procedure-parameters procedure)
           (list-of-delayed-args arguments env)
           (procedure-environment procedure))))
        (else (error "Unknown proc" procedure))))
```

7/40

## Lazy Evaluation – l-eval

- Most of the work is in l-apply; need to call it with:
  - actual value for the operator
  - just expressions for the operands
  - the environment...

```
(define (l-eval exp env)
  (cond ((self-evaluating? exp) exp)
        ...
        ((application? exp)
         (l-apply (actual-value (operator exp) env)
                  (operands exp)
                  env))
        (else (error "Unknown expression" exp))))
```

8/40

## Meval versus L-Eval

```
(define (meval exp env)
  (cond ((self-evaluating? exp) exp)
        ...
        ((cond? exp) (meval (cond->if exp) env))
        ((application? exp)
         (mapply (meval (operator exp) env)
                  (list-of-values (operands exp) env)))
        (else (error "Unknown expression type -- EVAL" exp))))

(define (l-eval exp env)
  (cond ((self-evaluating? exp) exp)
        ...
        ((cond? exp)
         (meval (cond->if exp) env))
        ((application? exp)
         (l-apply (actual-value (operator exp) env)
                  (operands exp)
                  env))
        (else (error "Unknown expression" exp))))
```

9/40

## Actual vs. Delayed Values

```
(define (actual-value exp env)
  (force-it (l-eval exp env)))

(define (list-of-arg-values exps env) Used when applying a
  (if (no-operands? exps) '() primitive procedure
      (cons (actual-value (first-operand exps) env)
            (list-of-arg-values (rest-operands exps)
                               env))))

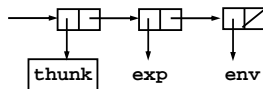
(define (list-of-delayed-args exps env) Used when applying a
  (if (no-operands? exps) '() compound procedure
      (cons (delay-it (first-operand exps) env)
            (list-of-delayed-args (rest-operands exps)
                                   env))))
```

10/40

## Representing Thunks

- *Abstractly* – a **thunk** is a "promise" to return a value when later needed ("forced")

- *Concretely* – our representation:



11/40

## Thunks – delay-it and force-it

```
(define (delay-it exp env) (list 'thunk exp env))
(define (thunk? obj) (tagged-list? obj 'thunk))
(define (thunk-exp thunk) (cadr thunk))
(define (thunk-env thunk) (caddr thunk))
```

```
(define (force-it obj)
  (cond ((thunk? obj)
        (actual-value (thunk-exp obj)
                      (thunk-env obj)))
        (else obj)))

(define (actual-value exp env)
  (force-it (l-eval exp env)))
```

12/40

## Memo-izing evaluation

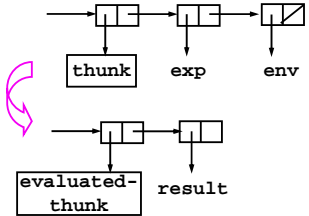
- In lazy evaluation, if we reuse an argument, have to reevaluate each time
- In normal evaluation, argument is evaluated once, and just referenced
- Can we keep track of values once we've obtained them, and avoid cost of reevaluation?

13/40

## Memo-izing Thunks

- *Idea*: once thunk `exp` has been evaluated, remember it
- If value is needed again, just return it rather than recompute

- *Concretely* – mutate a thunk into an evaluated-thunk



14/40

## Thunks – Memoizing Implementation

```
(define (evaluated-thunk? obj)
  (tagged-list? obj 'evaluated-thunk))
(define (thunk-value evaluated-thunk)
  (cadr evaluated-thunk))

(define (force-it obj)
  (cond ((thunk? obj)
        (let ((result (actual-value (thunk-exp obj)
                                   (thunk-env obj))))
          (set-car! obj 'evaluated-thunk)
          (set-car! (cdr obj) result)
          (set-cdr! (cdr obj) '())
          result))
        ((evaluated-thunk? obj) (thunk-value obj))
        (else obj))))
```

15/40

## Lazy Evaluation – other changes needed

- Example – need actual predicate value in conditional if...
 

```
(define (l-eval-if exp env)
  (if (true? (actual-value (if-predicate exp) env))
      (l-eval (if-consequent exp) env)
      (l-eval (if-alternative exp) env)))
```
- Example – don't need actual value in assignment...
 

```
(define (l-eval-assignment exp env)
  (set-variable-value!
   (assignment-variable exp)
   (l-eval (assignment-value exp) env)
   env)
  'ok)
```

16/40

## Laziness and Language Design

- We have a dilemma with lazy evaluation
  - Advantage: only do work when value actually needed
  - Disadvantages
    - not sure when expression will be evaluated; can be very big issue in a language with side effects
    - may evaluate same expression more than once
- Memoization doesn't fully resolve our dilemma
  - Advantage: Evaluate expression at most once
  - Disadvantage: What if we *want* evaluation on each use?
- Alternative approach: **give programmer control!**

17/40

## Variable Declarations: lazy and lazy-memo

- Handle lazy and lazy-memo extensions in an upward-compatible fashion.;

```
(lambda (a (b lazy) c (d lazy-memo)) ...)
```

- "a", "c" are normal variables (evaluated before procedure application)
- "b" is lazy; it gets (re)-evaluated each time its value is actually needed
- "d" is lazy-memo; it gets evaluated the first time its value is needed, and then that value is returned again any other time it is needed again.

18/40

## Syntax Extensions – Parameter Declarations

```
(define (first-variable var-decls) (car var-decls))
(define (rest-variables var-decls) (cdr var-decls))
(define declaration? pair?)

(define (parameter-name var-decl)
  (if (pair? var-decl) (car var-decl) var-decl))

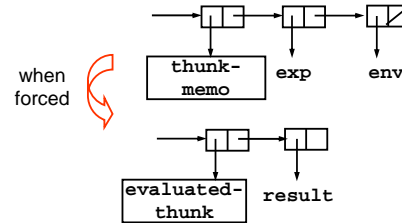
(define (lazy? var-decl)
  (and (pair? var-decl) (eq? 'lazy (cadr var-decl))))

(define (memo? var-decl)
  (and (pair? var-decl)
       (eq? 'lazy-memo (cadr var-decl))))
```

19/40

## Controllably Memo-izing Thunks

- **thunk** – never gets memoized
- **thunk-memo** – first eval is remembered
- **evaluated-thunk** – memoized-thunk that has already been evaluated



20/40

## A new version of delay-it

- Look at the variable declaration to do the right thing...

```
(define (delay-it decl exp env)
  (cond ((not (declaration? decl))
        (1-eval exp env))
        ((lazy? decl)
         (list 'thunk exp env))
        ((memo? decl)
         (list 'thunk-memo exp env))
        (else (error "unknown declaration:" decl))))
```

21/40

## Change to force-it

```
(define (force-it obj)
  (cond ((thunk? obj) ;eval, but don't remember it
        (actual-value (thunk-exp obj)
                      (thunk-env obj)))
        ((memoized-thunk? obj) ;eval and remember
         (let ((result)
               (actual-value (thunk-exp obj)
                             (thunk-env obj))))
           (set-car! obj 'evaluated-thunk)
           (set-car! (cdr obj) result)
           (set-cdr! (cdr obj) '())
           result))
        ((evaluated-thunk? obj) (thunk-value obj))
        (else obj)))
```

22/40

## Changes to l-apply

- Key: in l-apply, only delay "lazy" or "lazy-memo" params
  - make thunks for "lazy" parameters
  - make memoized-thunks for "lazy-memo" parameters

```
(define (l-apply procedure arguments env)
  (cond ((primitive-procedure? procedure)
        ...) ; as before; apply on list-of-arg-values
        ((compound-procedure? procedure)
         (1-eval-sequence
          (procedure-body procedure)
          (let ((params (procedure-parameters procedure)))
              (extend-environment
               (map parameter-name params)
               (list-of-delayed-args params arguments env)
               (procedure-environment procedure))))))
        (else (error "Unknown proc" procedure))))
```

23/40

## Deciding when to evaluate an argument...

- Process each variable declaration together with application subexpressions – delay as necessary:

```
(define (list-of-delayed-args var-decls exps env)
  (if (no-operands? exps)
      '()
      (cons (delay-it (first-variable var-decls)
                     (first-operand exps)
                     env)
            (list-of-delayed-args
             (rest-variables var-decls)
             (rest-operands exps)
             env))))
```

24/40

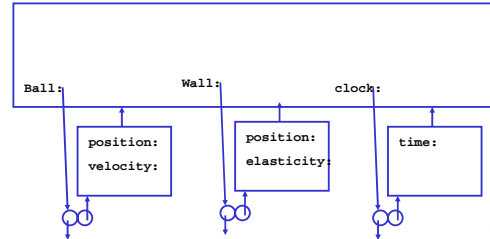
## Summary

- Lazy evaluation – control over evaluation models
  - Convert entire language to normal order
  - Upward compatible extension
    - lazy & lazy-memo parameter declarations

25/40

## Streams – a different way of structuring computation

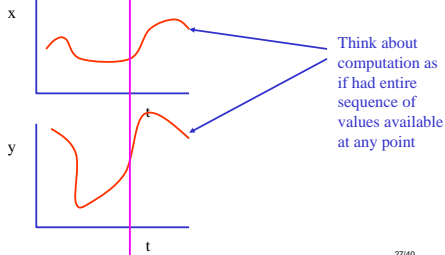
- Imagine simulating the motion of an object
  - Use state variables, clock, equations of motion to update
  - State of the simulation captured in instantaneous values of state variables



26/40

## Streams – a different way of structuring computation

- OR – have each object output a continuous stream of information
  - State of the simulation captured in the history (or stream) of values



27/40

## How do we use this new lazy evaluation?

- Our users could implement a *stream abstraction*:
 

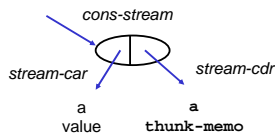
```
(define (cons-stream x (y lazy-memo))
  (lambda (msg)
    (cond ((eq? msg 'stream-car) x)
          ((eq? msg 'stream-cdr) y)
          (else (error "unknown stream msg" msg)))))

(define (stream-car s) (s 'stream-car))
(define (stream-cdr s) (s 'stream-cdr))
OR
(define (cons-stream x (y lazy-memo))
  (cons x y))
(define stream-car car)
(define stream-cdr cdr)
```

28/40

## Stream Object

- A pair-like object, except the cdr part is *lazy* (not evaluated until needed):



- Example
 

```
(define x (cons-stream 99 (/ 1 0)))
(stream-car x) => 99
(stream-cdr x) => error - divide by zero
```

29/40

## Decoupling computation from description

- Can separate order of events in computer from apparent order of events in procedure description

```
(list-ref
 (filter (lambda (x) (prime? x))
         (enumerate-interval 1 100000000))
 100)

(define (stream-interval a b)
  (if (> a b)
      the-empty-stream
      (cons-stream a (stream-interval (+ a 1) b))))

(stream-ref
 (stream-filter (lambda (x) (prime? x))
                (stream-interval 1 100000000))
 100)
```

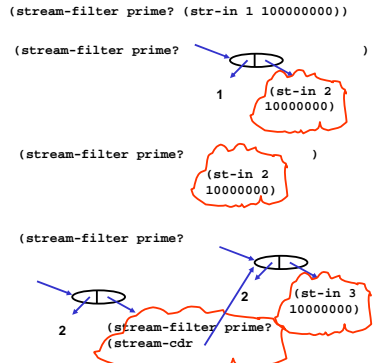
30/40

### Some details on stream procedures

```
(define (stream-filter pred str)
  (if (pred (stream-car str))
      (cons-stream (stream-car str)
                   (stream-filter pred
                                   (stream-cdr str)))
      (stream-filter pred
                    (stream-cdr str))))
```

31/40

### Decoupling order of evaluation

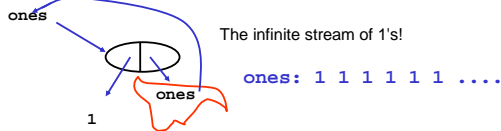


32/40

### Result: Infinite Data Structures!

- Some very interesting behavior
 

```
(define ones (cons-stream 1 ones))
(stream-car (stream-cdr ones)) => 1
```



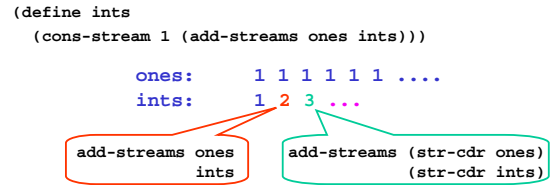
- Compare:
 

```
(define ones (cons 1 ones)) => error, ones undefined
```

33/40

### Finite list procs turn into infinite stream procs

```
(define (add-streams s1 s2)
  (cond ((null? s1) '())
        ((null? s2) '())
        (else (cons-stream
                (+ (stream-car s1) (stream-car s2))
                (add-streams (stream-cdr s1)
                              (stream-cdr s2))))))
```



34/40

### Finding all the primes

2	3	X	5	X	7	X	9	X	X
11	X	13	X	X	X	17	X	19	X
X	X	23	X	X	X	X	X	29	X
31	X	X	X	X	X	37	X	X	X
41	X	43	X	X	X	47	X	X	X
X	X	53	X	X	X	X	X	59	X
61	X	X	X	X	X	67	X	X	X
71	X	73	X	X	X	X	X	79	X
X	X	83	X	X	X	X	X	89	X
X	X	X	X	X	X	97	X	X	X

35/40

### Building a sieve?

```
(define (sieve str)
  (cons-stream
    (stream-car str)
    (sieve (stream-filter
            (lambda (x)
              (not (divisible? x (stream-car str))))
            (stream-cdr str)))))

(define primes
  (sieve (stream-cdr ints)))

(2 (sieve (filter ints 2)))

(2 3 (sieve (filter
            (sieve (filter ints 2))
            3)))
```

36/40

### Streams Programming

- Signal processing:
- Streams model:

37/40

### Integration as an example

```

(define (integral integrand init dt)
  (define int
    (cons-stream
      init
      (add-streams (stream-scale dt integrand)
                   int)))
  int)

(integral ones 0 2)
=> 0 2 4 6 8
Ones: 1 1 1 1 1
Scale 2 2 2 2 2

```

38/40

### Summary

- We can control when arguments are evaluated
  - By making a lazy evaluator
  - By changing the evaluator to allow specification of arguments
- Changing the evaluator requires a small amount of work but dramatically shifts the behavior of the system
  - Applicative order versus Normal order
- Using a lazy evaluator lets us separate the apparent order of computation inherent in a problem from the actual order of evaluation inside the machine

39/40

### Stages of an interpreter

input to each stage

**Lexical analyzer** "(average 4 (+ 5 5))"

**Parser** ( average 4 ( + 5 5 ) )

**Evaluator** → [ ] → [ ] → [ ]

**Environment** symbol average, symbol +, 4, 5, 5

**Printer** 7

40/40